Mixing Hadoop and HPC Workloads on Parallel Filesystems

Esteban Molina-Estolano[1], Maya Gokhale[2], Carlos Maltzahn[1], John May[2], John Bent[3], and Scott Brandt[1]

[1]UC Santa Cruz

[2]Lawrence Livermore National Laboratory

[3]Los Alamos National Laboratory

ABSTRACT

MapReduce-tailored distributed filesystems—such as HDFS for Hadoop MapReduce—and parallel high-performance computing filesystems are tailored for considerably different workloads. The purpose of our work is to examine the performance of each filesystem when both sorts of workload run on it concurrently.

We examine two workloads on two filesystems. For the HPC workload, we use the IOR checkpointing benchmark and the Parallel Virtual File System, Version 2 (PVFS); for Hadoop, we use an HTTP attack classifier and the CloudStore filesystem. We analyze the performance of each file system when it concurrently runs its "native" workload as well as the non-native workload.

1. INTRODUCTION

We examine the possibility of using the same filesystem installation for both Hadoop MapReduce workloads and high-performance computing workloads. We examine the performance of *mixed workloads*: a Hadoop workload and an HPC checkpointing workload running concurrently on the same filesystem.

MapReduce is an algorithm and execution strategy used for processing very large data sets on clusters of commodity machines holding the data. Analysis jobs that fit the MapReduce style are written for a particular MapReduce engine; the engine parallelizes the work across a cluster, and typically provides fault tolerance.

There are multiple implementations of MapReduce, with different filesystem requirements. Google's MapReduce implementation[4] heavily uses atomic appends to files, a feature supported by the Google File System[7]. In this work, we use a different implementation: Apache's Hadoop MapReduce[1], a Java-based implementation that uses write-once files instead of atomic appends. Both filesystems use replication, as they are built to run MapReduce on unreliable commodity machines.

On the other hand, parallel high-performance computing (HPC) filesystems are designed for more general-purpose workloads in which data is transmitted between storage system and compute cluster over an interconnection network. At a site where both types of workloads are of interest, it would be useful to support both workloads concurrently with a single filesystem installation. For instance, consider a large filesystem used for simulation work: the same parallel filesystem could save checkpoints for the currently running simulation, while running analysis jobs on the raw results from a previous simulation.

Furthermore, the greater fault tolerance of the chunk replication mechanism in MapReduce-tailored filesystems would be useful for large HPC clusters; we know of no production-quality parallel filesystem that offers declustered replication on unreliable commodity hardware. (Ceph[11] offers declustered replication on unreliable commodity hardware, but it is still alpha-quality.)

Tantisiriroj et al. [10] at CMU consider the related problem of running Hadoop workloads on parallel HPC parallel filesystems, They develop a shim to allow Hadoop to run on the Parallel Virtual File System, Version 2 (PVFS), and show comparable performance with Hadoop's HDFS distributed filesystem.

We examine the mixed workloads under two filesystems: one suited for each type of workload. We use PVFS, which is suited for HPC workloads, using the aforementioned shim; and we use CloudStore, which is suited for Hadoop workloads. We also add infrastructure to Hadoop and to our chosen HPC benchmark, the Sequoia IOR benchmark, to collect traces of parallel file I/O for both workloads; and measure the trace overhead.

2. THE FILESYSTEMS

Most parallel filesystems share a basic architecture, consisting of a metadata server (called a *nameserver* in HDFS and CloudStore), which handles operations such as creating and opening files; and multiple storage nodes, which store chunks of data. Individual files are split into chunks, which are stored on different nodes according to a placement policy. A client contacts the metadata server to open or create a file. To read or write, it uses the placement policy to locate the relevant chunks, and makes read and write requests directly with the storage nodes hosting the chunks. Clients may run on the same cluster as the filesystem, or on a different cluster entirely.

Hadoop workloads and HPC workloads are each supported by multiple filesystems. Our filesystem choices for this work are largely determined by the requirement that each filesystem host both types of workloads. This is complicated by interface and file semantics differences: while HPC checkpointing workloads generally use a POSIX-like interface and support shared writes, Hadoop uses its own custom filesystem API, tailored to single-writer write-once workloads.

When selecting a Hadoop-native filesystem that could also support HPC checkpointing workloads, we found that Hadoop's HDFS

was unsuitable: HDFS does not support shared writes to one file by multiple clients, a common checkpointing pattern. However, CloudStore[8], another Hadoop-tailored filesystem, does support shared writes. Choosing an HPC-native filesystem to run Hadoop workloads was straightforward: since the writers of the PVFS Hadoop shim were kind enough to provide us a prerelease version, we used PVFS. We briefly describe the two filesystems below, and discuss their suitability of each for the non-native workload.

2.1 CloudStore

The CloudStore filesystem, formerly called KosmosFS (KFS), is a distributed filesystem similar to HDFS, and tailored for Hadoop workloads.

In typical HPC filesystems, the clients performing the computation are on a separate cluster from the storage cluster; the two clusters communicate via high-speed interconnects. By contrast, typical MapReduce installations use the same set of nodes for both storage and computation. Combined with a large chunk size, this provides an opportunity to exploit locality: placing clients on the nodes that already hold their desired data reduces network bandwidth. By default, files in CloudStore are written in 64 MB chunks, with three replicas of each chunk for reliability. The selection of chunkservers to store replicas is done as follows: one replica is stored on the chunkserver hosted on the client machine itself, incurring no network bandwidth; one replica is stored on a different chunkserver, on the same rack; and one replica is stored on a chunkserver on a different rack. Hadoop partitions the job into Map tasks, and attempts to place the Map tasks near their data.

2.2 PVFS

PVFS[3], a parallel filesystem suited to high-performance computing workloads, uses a considerably different data layout. Files are split into small chunks (64 KB by default); a file's first chunk is stored on a randomly-selected storage node, and subsequent chunks of the file are stored across nodes in a round-robin fashion. PVFS does not replicate data across nodes. Typically, PVFS clients are on nodes separate from the storage cluster.

We use the PVFS Hadoop shim to run Hadoop on PVFS. Typically for Hadoop, but atypically for PVFS, we run Hadoop and PVFS on the same set of nodes. The shim adapts PVFS to better suit Hadoop workloads: it adds readahead buffering and replication, and exposes the data layout to let Hadoop exploit locality. As in [10], we also set PVFS to use 64 MB chunks: since Hadoop takes the chunk size into account when partitioning a job, the small default chunk size of PVFS yields an excessively large number of tiny Map tasks, slowing performance by an order of magnitude. Tantisiriroj et al. found PVFS with the shim to be comparable in performance to HDFS for Hadoop MapReduce-workloads, except for write-heavy workloads.

3. WORKLOADS

3.1 IOR benchmark

As a representative HPC workload, we use the IOR parallel I/O benchmark from LLNL's ASC Sequoia set of HPC benchmarks[2], which supports various access patterns.We use IOR to simulate HPC checkpointing, where each client in a running simulation periodically dumps its state to disk. There are two typical checkpointing workloads: one an N-N workload, where each client writes its state to a separate file; and the other an N-1 workload, where all clients write their state to the same file. An N-1 workload may be nonstrided, with each client using a separate, contiguous region of the file; or strided, where the clients write their state to the file in an interleaved fashion. Though HPC checkpointing occurs periodically throughout a simulation, we do not simulate this periodicity; each IOR experiment simulates a single checkpoint.

IOR has support for several filesystem APIs, but CloudStore is not among them. To run IOR unmodified on CloudStore, we would need to mount CloudStore as a regular Linux filesystem. In Linux, there are two ways to do this: using a kernel module for the filesystem, or using a FUSE module. However, CloudStore has no Linux kernel module; and while it does have a FUSE module, using FUSE adds overhead. To avoid incurring the overhead of FUSE, we avoided mounting CloudStore. We instead modified IOR to use CloudStore's POSIX-like C++ library directly. We make no effort to tune CloudStore to improve its performance with IOR; this is in contrast to the Hadoop PVFS shim, which makes several adaptations to improve Hadoop performance on PVFS.

3.2 Hadoop TFIDF workload

As our Hadoop workload, we use a Hadoop implementation of an HTTP attack classifier, which detects malicious code in incoming HTTP requests. The classifier uses a Term Frequency-Inverse Document Frequency (TFIDF) metric[6]. For brevity, we refer to this as the *TFIDF workload*. The workload partitions a large set of HTTP requests among a number of Map tasks; each Map task reads a previously generated 69 MB model file, and uses it to classify its share of the dataset.

Unlike many MapReduce workloads, this workload lacks a Reduce phase; each Map task generates one results file, and the set of results files comprises the result of the job. We expect that a Reduce phase would incur more network bandwidth. The Map tasks send their intermediate results to one or a few nodes; thus, the opportunity to exploit locality is lower than in the Map phase. Unfortunately, we cannot yet fully trace Hadoop jobs with a Reduce phase.

4. TRACING INFRASTRUCTURE

We considered existing tracing methods, such as strace and Pianola[9], but decided to write our own tracing mechanisms for IOR and Hadoop.

To trace IOR, we opted to add a shim layer in the IOR code itself, to record a trace of all the read and write calls from each client. The low complexity of the IOR code made this straightforward; we simply wrapped the file I/O calls in the code, and sent trace information to `stdout`. Since we linked CloudStore support directly into IOR, bypassing the Linux filesystem interface, system call tracing tools, such as strace and Pianola, would not have captured CloudStore I/O.

Tracing Hadoop was more involved. As with IOR on CloudStore, system call tracing tools were unsuitable, as Hadoop does not use the Linux filesystem interface when performing parallel I/O. Hadoop exposes an abstract filesystem interface in Java, via the abstract `FileSystem` class, which handles metadata operations; and the `FSDataInputStream` and `FSDataOutputStream` classes, which represent files open for reading and writing, respectively. The write-once-read-many API differs from POSIX: notably, files are written only on creation, by a single writer, with no seeking supported. Particular storage systems, such as HDFS, CloudStore, and Amazon's S3, are supported in Hadoop via implementations of these classes. The default filesystem to use in Hadoop is specified in a configuration file by a URI. For instance, to use PVFS, we use a URI of the form `pvfs2://[server]:[port]`, pointing to a PVFS server in the cluster. Given this URI, Hadoop knows (via configuration settings) to instantiate the `PVFS2FileSystem` class.

To trace Hadoop I/O, we implement versions of the three classes designed to trace another filesystem. For instance, to trace PVFS, we use a URI of the form `tfs://pvfs2-[server]:[port]`. Hadoop instantiates our class, `TracerFileSystem`, which itself instantiates `PVFS2FileSystem`. The tracer captures all metadata calls, logs them using Hadoop's existing logging system, and forwards them to PVFS. Unfortunately, because of issues with `final` methods and the Hadoop class hierarchy, we could not use the same method to wrap the actual input and output streams; we were forced to modify the PVFS and CloudStore FileSystem implementations to wrap the streams and trace read and write operations. These wrappers likewise use Hadoop's existing logging system. The trace output resembles that of strace, giving the timestamp, the operation and its parameters, the return value (if any), and the execution time.

Tracing the raw read and write calls in Hadoop produced an excessive volume of trace data, severely degrading performance. Reading the classification model caused much of the problem: each Map task read it a few bytes at a time. To reduce the amount of trace output, we placed a read buffer with some readahead between the read calls and the tracing mechanism; and placed a write buffer between the write calls and the tracing mechanism. (The Hadoop PVFS shim already has suitable buffers; we merely placed the trace mechanism on the opposite side of the buffers from the I/O calls.) We left these buffers in place even with tracing turned off.

Hadoop MapReduce jobs, in general, stores intermediate results in local temp directories, off the parallel filesystem, and sends them to the appropriate Reduce tasks via HTTP. Our tracing mechanism cannot currently trace this I/O, as it only traces I/O on the parallel filesystem. This is not a problem for the TFIDF workload, which skips the reduce phase and writes Map results directly to the parallel filesystem. However, this omission must be filled for our tracing mechanism to completely trace Hadoop jobs that do include a Reduce phase; we plan to address it in future work.

With any tracing mechanism, performance effects are a concern. The tracing mechanisms for both IOR and Hadoop store trace data on an NFS share separate from the experimental cluster. This avoids contention with the job for disk I/O, but not for CPU and network bandwidth. The results in Section 5 come from runs with tracing turned off; to measure overhead, we also ran each experiment with tracing turned on.

Since a faster-performing experiment will, in general, produce trace data more quickly, we examine our fastest-performing runs for their rate of trace data generation. We found that our fastest IOR runs produced trace data at about 2 MB/s; and our fastest TFIDF runs produced trace data at about 8 MB/s. We attribute the larger trace sizes in TFIDF to a larger number of smaller I/O operations, and to a more verbose trace format.[1]

We measure tracing overhead by comparing performance results from traced runs with those for the equivalent untraced runs. For our baseline (non-mixed) experiments, the tracing overhead was small. For IOR, most traced results were within 0.3 MB/s of the equivalent untraced result. For TFIDF, tracing added 5.4% on average to the runtime under PVFS, and 2.6% to the runtime under CloudStore.

The overhead for the mixed workloads, with both workloads running and being traced at once, was greater in most cases than the overhead for the corresponding baseline workload. Tracing overhead for IOR was 1–2 MB/s for both filesystems. For TFIDF, tracing overhead increased the runtime by 11% with Cloudstore. The exception was TFIDF on PVFS: the runtime increased by less than 1.5%, lower than the 5.4% overhead in the baseline run.

[1]Hadoop adds extra information to log entries; and we included filenames in trace entries, to improve readability.

5. EXPERIMENTS

We ran our tests on 19 nodes of the `tuson` cluster at LLNL. Each node has a 2-core 2.4 GHz Intel Xeon (32-bit) CPU, 4 GB of RAM, a 120 GB disk, and a Gigabit Ethernet connection. We ran each experiment 5 times; we measured bandwidth once per run, by dividing the total workload size by the runtime, and we give the average of these bandwidths for each experiment.

We ran three different baseline workloads on each filesystem, for a total of six baseline experiments. The first workload is an N-N IOR checkpointing workload with a total write size of 4.75 GB; each client writes 256 MB to its own file, using 256 KB write operations. The second workload is the same size, but is an N-1 strided workload with a chunk size of 64 MB. All clients write to the same file, writing in interleaved 64 MB chunks. Again, each client writes a total of 256 MB, using 256 KB write operations. The third workload is the TFIDF workload, classifying a 7.2 GB dataset. This dataset is made of 64 copies of the dataset from the ECML/PKDD 2007 Discovery Challenge[5].

We ran four mixed workload experiments: two different mixed workloads on each filesystem. One mixed workload runs IOR N-1 and TFIDF concurrently; the other runs IOR N-N and TFIDF concurrently. Since we measured the performance of both workloads, the different running times were a concern: TFIDF runs for considerably longer than either of the IOR runs. To compensate, we increased the size of the IOR run in each mixed workload so that the IOR and TFIDF runtimes matched; each one finished within a few seconds of the other.

5.1 Results

The behavior of Hadoop's scheduler is a factor in our results. Among its tasks, the job scheduler decides which node to place each Map task on. Each TFIDF run used 114 Map tasks. In different experiments, the Hadoop scheduler made different numbers of them *data-local* tasks: tasks running on the same node that hold their input data, thereby reducing network traffic.

We first present the results for the experiments on CloudStore. The left-hand side of Figure 1 shows the performance of the TFIDF workload on its own, with IOR N-1, and with IOR N-N; IOR runs here with an output size of 4.75 GB. We found that TFIDF incurred a performance hit of about a third under the mixed workload. The left-hand side of Figure 2 shows the performance of the IOR workload, including the mixed-workload IOR results from the same experiments that produced the mixed-workload CloudStore TFIDF results: the N-1 workload is slowed down by about a third, and the N-N workload is slowed by about a fourth.

That the slowdown is always less than half suggests that neither workload is using the CloudStore filesystem near its full capacity. Since the IOR benchmark does nothing but write data, this suggests that CloudStore is not well-suited out of the box for checkpointing workloads. This is reinforced by the PVFS results we present below; for nearly every workload combination, IOR runs faster on PVFS than on CloudStore.

For the baseline CloudStore TFIDF experiment, the Hadoop scheduler made none of the Map tasks data-local; for the mixed CloudStore experiments, made an average of 95 of 114 Map tasks data-local.

We next present the results for the same workloads, run on PVFS. The right-hand side of Figure 1 shows the performance of TFIDF. Surprisingly, on PVFS, TFIDF is only marginally slowed down when IOR is running concurrently. On the other hand, both IOR N-1 and IOR N-N are slowed down by half, as shown in the right hand side of Figure 2.

Figure 3 shows a different view of the performance results. Us-

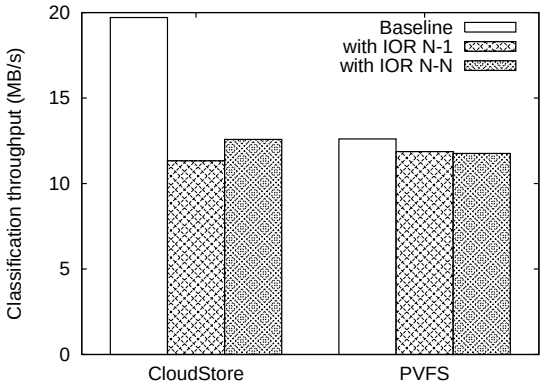

Figure 1: We show the performance results for the TFIDF workload on both CloudStore and PVFS. We show results for TFIDF running on its own; concurrently with IOR N-1; and concurrently with IOR N-N.

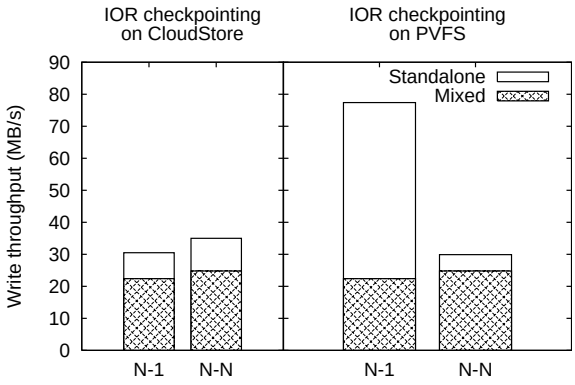

Figure 2: We show the performance of the IOR workload running on each filesystem. We show results for IOR N-N and IOR N-1, both alone and concurrently with TFIDF.

ing the results of the baseline experiments, we compute the runtime that each mixed-workload experiment would have taken, had the jobs been run sequentially instead of concurrently; and plot those with the actual mixed-workload runtimes. In all cases, the mixed-workload runtime is lower than the sequential runtime.

For the PVFS TFIDF experiments, the Hadoop scheduler's decisions were almost exactly the reverse of its decisions for CloudStore. The baseline PVFS experiment had an average of 92 data-local Map tasks; and the mixed PVFS experiments had no data-local Map tasks. We hypothesize that much of the baseline performance difference for TFIDF between PVFS and CloudStore comes from the greater locality in PVFS.

As future work, we will examine the behavior of the Hadoop job scheduler to determine the cause of these differences. We will modify the scheduler to take control of the Map task placement, and rerun our benchmarks with locality forced high, and locality forced low; this will allow us to measure the performance effects of locality in TFIDF, for both standalone and mixed workloads.

6. CONCLUSIONS AND FUTURE WORK

We examined the performance of mixed workloads running on the same filesystem: a Hadoop workload with an HPC checkpointing workload. We measured performance on two filesystems, one tailored to each workload: the Hadoop-friendly CloudStore filesystem, and the HPC-oriented PVFS filesystem. We also developed a mechanism to record file I/O traces from both workloads, and found it to have a small overhead in most cases.

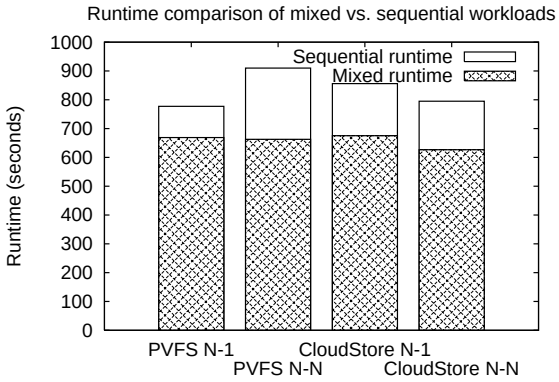

Figure 3: We show the runtime of each mixed workload, compared to the total runtime (computed from baseline experiments) if the two parts of the workload run separately. In each case, the total runtime is lower for the mixed workload.

We found that the TFIDF Hadoop workload, when run alone, ran considerably more slowly on PVFS than on CloudStore. We found that TFIDF, running on PVFS, was scarcely slowed down by a concurrent running checkpointing workload; but in all other cases, jobs were slowed down substantially when run as part of a mixed workload. We also discovered that the Hadoop job scheduler made different placement decisions for TFIDF Map tasks for the mixed workload runs, when compared to identically configured standalone TFIDF runs.

We have several promising avenues for future work. We will determine why the Hadoop scheduler chooses to exploit locality in some cases and not others. We plan to run experiments on larger numbers of nodes; this will allow us to examine how scalability is affected by mixed workloads. We also plan to improve our Hadoop tracing mechanism by reducing the size of trace output to improve performance, and by adding a way to trace the file I/O that Hadoop performs in local temporary directories. Then, we will be able to run and properly trace Hadoop jobs with a Reduce phase. We would also like to improve the performance of checkpointing workloads on CloudStore.

Acknowledgments.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was funded by LLNL LDRD project 07-ERD-063. We thank Wittawat Tantisiriroj, Swapnil Patil, Garth Gibson, and Greg Ganger for providing us a pre-release version of their Hadoop PVFS shim. We also thank Sriram S. Rao and the kosmosfs-devel mailing list for assistance with CloudStore. This work was also supported in part by the Department of Energy under award DE-FC02-06ER25768 (PDSI) and the LANL/UCSC Institute for Scalable Scientific Data Management. We thank the members of the UCSC Systems Research Lab whose advice helped guide this research.

7. REFERENCES

[1] Apache. Welcome to apache hadoop! http://hadoop.apache.org/.

[2] LLNL ASC. Asc sequoia benchmark codes. https://asc.llnl.gov/sequoia/benchmarks/.

[3] Philip H. Carns, Iii, Robert B. Ross, and Rajeev Thakur. Pvfs: a parallel file system for linux clusters. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, page 28, Berkeley, CA, USA, 2000. USENIX Association.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[5] ECML. Ecml/pkdd discovery challenge. http://www.ecmlpkdd2007.org/.

[6] Brian Gallagher and Tina Eliassi-Rad. Classification of http attacks: A study on the ecml/pkdd 2007 discovery challenge, 2009.

[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[8] Kosmos. About cloudstore. kosmosfs.sourceforge.net/about.html.

[9] John May. Pianola: A script-based i/o benchmark. In *Proceedings of the 2008 ACM Petascale Data Storage Workshop (PDSW 08)*, November 2008.

[10] Wittawat Tantisiriroj, Swapnil Patil, and Garth Gibson. Data-intensive file systems for internet services: A rose by any other name..., 2008.

[11] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*, volume 7. USENIX, November 2006.